(2)

IDA PAPER P-2899

COMPARING ADA AND FORTRAN LINES OF CODE: SOME EXPERIMENTAL RESULTS

Thomas P. Frazier
John W. Bailey
Melissa L. Young

November 1993

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

IDA Log No. HQ 93-044552

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this publication was conducted under IDA's Independent Research Program. Its publication does not imply endorsement by the Department of Defense, or any other Government agency, nor should the contents be construed as reflecting the official position of any Government agency.

UNCLASSIFIED

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1993	Final Report, Apr 1993 – Nov 1993

4. TITLE AND SUBTITLE
Comparing Ada and FORTRAN Lines of Code:
Some Experimental Results

5. FUNDING NUMBERS
IDA CRP 9000-729

6. AUTHOR(S)
Thomas P. Frazier, John W. Bailey, and Melissa L. Young

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses
1801 N. Beauregard Street
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2899

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
FFRDC Programs
5109 Leesburg Pike, Suite 317
Falls Church, VA 22041

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12B. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper presents the results of an experiment in rewriting four FORTRAN programs in the Ada computer language. We counted and compared the numbers of lines of code in both languages. We found Ada programs to be larger than their functionally equivalent FORTRAN counterparts. However, we observed that the overhead for Ada diminishes as the program size is increased. Our limited data suggested that there may even be a cross-over point beyond which the size of an Ada program is smaller than a functionally equivalent FORTRAN program.

14. SUBJECT TERMS
FORTRAN, Ada Programming Language, Computer Programs, Lines of Code

15. NUMBER OF PAGES
43

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

UNCLASSIFIED

IDA PAPER P-2899

COMPARING ADA AND FORTRAN LINES OF CODE: SOME EXPERIMENTAL RESULTS

Thomas P. Frazier
John W. Bailey
Melissa L. Young

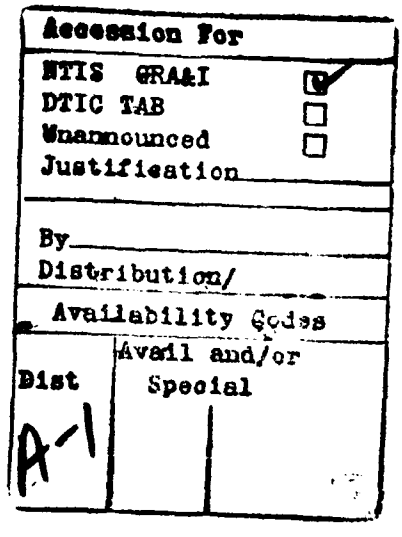
November 1993

INSTITUTE FOR DEFENSE ANALYSES
IDA Independent Research Program

PREFACE

This paper was prepared by the Institute for Defense Analyses (IDA) under the IDA Independent Research Program. The objective of the research was to compare the relative sizes of functionally equivalent programs written in the Ada and FORTRAN computer languages.

This paper was reviewed by Bruce N. Angier and D. Graham McBryde.



CONTENTS

I. Introduction I-1
A. Background I-1
B. Objective I-2
II. Approach II-1
A. Test Programs II-1
B. Ada and FORTRAN Formatting and Style II-2
C. Code-Counting Conventions: Sizing Issues II-3
III. Results III-1
A. PSS and LSS Counts III-1
B. Declarations III-3
C. Statements III-5
D. Economies of Scale in the Ada Language III-6
IV. Conclusions and Future Research IV-1
Appendix A: Ada and FORTRAN Code for Example Programs A-1
References B-1
Abbreviations C-1

FIGURE

III-1. Break-Even Size III-6

TABLES

II-1. Comparison of Ada and FORTRAN "if...then" Statements II-4
III-1. Lines-of-Code Count for Four Programs III-1
III-2. Executable and Declaration Code Count III-8
III-3. Declarations Versus Statements III-9
III-4. Estimates of Error III-9

I. INTRODUCTION

A. BACKGROUND

The introduction of the Ada computer programming language in the early 1980s has been cited as one of the major weapons in the fight to reduce the proliferation of computer languages and to control the cost of software in the Department of Defense (DoD). The features of the Ada language were carefully chosen to enable good engineering practices and structure to be imposed during the development and maintenance of computer software. However, the use of these structural and engineering features presents new problems for the cost analysts responsible for estimating either the size or the cost of software systems to be developed in Ada.

Most software cost-estimating models in use today assume that the cost of developing a computer program is a function of the size of the program (plus other representations or measures of the complexity of the program), the skills and experience of the programmers, and other factors that affect cost. Typically, these cost models use lines of code as a representation of the size of a software program or project. For example, Reference [1] employs the following relationship between effort to develop software and the size of the code

$$E = \alpha(KLOC)^{\beta}\prod_{i}^{n} m_i, \tag{1}$$

where E is defined to be the staff-months of development effort, α and β are the parameters that have been previously estimated, $KLOC$ is defined as thousands of source lines of delivered code, and m_i are multipliers or cost drivers that account for differences in software product attributes, computer attributes, personnel attributes and project attributes.

The use of a line of code as a unit of measure is appropriate and effective when dealing with line-oriented languages such as FORTRAN or assembly languages. However, several problems arise when applying a FORTRAN-specific or line-oriented cost model to software being developed in Ada.

First, instead of being line-oriented, Ada is block-oriented, which means its statements and declarations can span several lines or be nested within one another. This

implies that, instead of simply counting carriage returns, a special Ada-specific way of counting the effective number of lines in an Ada program is needed. Further, even given a way of measuring the size of an Ada program by some method of line counting, there is no assurance that a line of Ada by this definition will capture the same amount of function as a line of FORTRAN. This means that two functionally equivalent programs in the two languages might be considerably different in size, as measured by lines of code. Finally, there is no assurance that the cost to develop a line of Ada by this definition will be the same as the cost to develop a line of FORTRAN.

B. OBJECTIVE

This paper addresses the functional size issues but not the programming effort issues raised when comparing the sizes of Ada and FORTRAN programs.[1] However, there are currently no standard rules for normalizing the sizes of Ada developments and FORTRAN developments with respect to the functionality delivered.

Information about the relative sizes of functionally equivalent programs is needed by any organization that is considering a transition to the use of Ada in application areas in which they have previous experience in FORTRAN. The reasoning is that such an organization would be able to estimate the size of a programming job if it were developed in FORTRAN. However, it would have no way of knowing whether an Ada solution would be more or fewer lines of code. What is needed is the added knowledge about how large an Ada solution to a problem will be, given an estimate of size for a FORTRAN solution. This knowledge will allow FORTRAN organizations to "bootstrap" their software cost-estimating capabilities to include developments in the Ada language. Eventually, the need for this stop-gap technique will be eliminated by first-hand experience with Ada.

The focus of this study can be expressed in algebraic terms. The relationship between effort and size in line-oriented languages such as FORTRAN has been studied extensively by software engineers and cost analysts [3] and can be represented by equation (2),

$$E_F = \alpha(KLOC_F)^{\beta}\prod_i^n m_i, \qquad (2)$$

---

1 The programming effort issues can be addressed by observing the cost required to develop Ada programs of various sizes. There are several databases containing observations of productivity on Ada projects. One of the best examples is the work done at the MITRE Corporation and reported in Reference [2].

where the subscript F denotes FORTRAN. Researchers are learning about the relationship between effort and size in block-oriented languages such as Ada as represented by equation (3),

$$E_A = \delta(KLOC_A)^\gamma \prod_i^n m_i, \tag{3}$$

where the subscript A denotes Ada.

What is less known is the relationship between the size of an Ada program and a functionally equivalent FORTRAN program, or

$$KLOC_A = f(KLOC_F). \tag{4}$$

We focus on FORTRAN because the impetus for this research stems from the Institute for Defense Analyses' work for the Strategic Defense Initiative Organization (SDIO). Space systems have historically employed FORTRAN for both the ground segment software and the software embedded in the spacecraft or satellite itself. Cost-estimating relationships using FORTRAN lines of code have been the rule. However, the SDIO plans to field space systems software that will be predominately written in Ada. By determining the differences in size between functionally equivalent FORTRAN and Ada programs, this study will further our understanding of how traditional FORTRAN cost- and size-estimating models will have to be adjusted to handle the Ada language. In addition, by understanding the differences in size between functionally equivalent FORTRAN and Ada programs, we can estimate the error incurred by cost analysts when they simply use Ada and FORTRAN lines-of-code counts interchangeably.

II. APPROACH

In order to compare the sizes of functionally equivalent Ada and FORTRAN programs, we devised a simple experimental procedure. The procedure involved rewriting standard FORTRAN programs and routines using the Ada language. First, we developed an Ada solution for each program using the features of Ada as appropriate, such as packages and user-defined types. Because programming style can affect program size, we also wrote both terse and verbose versions of each of the FORTRAN and Ada programs. This yielded six functionally equivalent versions of each algorithm studied—three in FORTRAN and three in Ada. We then selected two established definitions for an Ada line of code and compared the number of Ada lines of code in these new programs to the number of lines in the original FORTRAN programs.

This chapter describes the test programs we selected and the formatting and code-counting conventions we employed.

A. TEST PROGRAMS

A total of four FORTRAN routines were used in the experiment. Three FORTRAN routines and their drivers were taken from *Numerical Recipes in FORTRAN* [4]. The National Aeronautics and Space Administration (NASA) Software Engineering Library (SEL) supplied the fourth FORTRAN program [5], along with an Ada translation, which we adapted for our basic Ada version of the program. Terse versions of the FORTRAN routines were devised by taking "shortcuts," such as allowing implicit declarations and eliminating certain unnecessary statements, such as format statements and continue statements. Verbose versions were devised by separately declaring variables, adding explicit format statements, and adding other optional statements to improve clarity. The terse versions of the Ada routines were devised by allowing multiple variables to appear in a single declaration and by using only positional parameter associations. The verbose versions were devised by separately declaring all variables and by using named parameter associations. By having a terse, normal, and verbose version of each algorithm in each language, we were able to obtain a useful picture of how the range of possible program sizes for a given function would differ in the two languages.

The three routines selected from *Numerical Recipes* were:

- Quicksort—a sorting routine that uses a "partition-exchange" sorting method.
- Fast Fourier Transform (FFT)—a computational algorithm that relates physical processes defined either in the time domain or frequency domain.
- Moments of a Distribution—a statistical routine that computes the moments (e.g., mean, variance, kurtosis) of a given distribution.

A fourth routine, an orbit propagator provided by the NASA/SEL, computes the orbital position of an earth satellite. These four were selected because they cover a range of computational applications likely to be used in space systems, and the algorithms involved are well known and widely used.

B. ADA AND FORTRAN FORMATTING AND STYLE

For comparisons we used two methods to measure the size of each of the Ada and FORTRAN subprograms and their drivers. Method 1 involves adopting a specific style for the formatting of the code and then simply counting the number of non-comment, non-blank lines in the file containing the code. The most complete definition we found for Ada formatting and style was "Ada Quality and Style: Guidelines for Professional Programmers" [6]. Except when deliberately employing either a verbose or a terse format, we adopted those rules of style for the examples of Ada used in this report. For the style of the FORTRAN examples, we followed the conventions detailed in *American National Standard Programming Language FORTRAN* (ANSI FORTRAN) [7]. This standard was adopted by the DoD in 1978.

Method 2 is a count of the number of source statements that appear in the code. Because this method measures the number of logical statements it is not sensitive to the number of physical lines a statement occupies. It is therefore not sensitive to formatting, comments or blank lines. Because of the multiple declaration option in Ada, this method is still somewhat sensitive to programming style, however. The specific declaration and statement counting rules we followed for both methods are described in "Code Counting Rules and Category Definitions/Relationships" [8]. To be consistent with the terms used in that report, we call Method 1 the physical source statement count, or the PSS count, and Method 2 the logical source statement count, or the LSS count.[2]

---

[2] The definitions of PSS and LSS are identical to the definitions in [9].

Although Reference [8] also discusses how to count comments in each language, we chose to ignore all comments and blank lines when measuring the sizes of our examples for this study. Also, we adopted the definition for a source statement to mean any programming instruction. In other words, all Ada declarations, statements, and pragmas are counted as source statements. In FORTRAN a source statement can be an executable statement, a data declaration, or a compiler directive.

C. CODE-COUNTING CONVENTIONS: SIZING ISSUES

The two selected methods for measuring program size are each compromises between the amount of the information captured by a size measure and the complexity of taking the measurement. The PSS method requires the program to be first formatted according to a set of rules and then simply counts the number of carriage returns in the code, excluding blank lines and comments. This approach either requires that a particular style be followed by the code developers or that a formatter (or "pretty printer") be used before any line counting is done. The LSS method defines a method for counting syntactic units rather than counting lines at all, so that the formatting of the code is immaterial. This LSS approach can be useful when reporting size outside of an individual development organization where styles and formatting rules may differ. However, it requires the additional complexity of processing or parsing the code in order to obtain the size count automatically.

Each statement, declaration, or pragma in Ada terminates with a semicolon (";"). Semicolons are also used to separate formal subprogram and entry parameters. Computing LSS means counting the semicolons except when they appear in (1) comments, (2) character literals, and (3) string literals. We decided to count the semicolons in formal parameter lists because formal parameters are, in effect, declarations. Although this count always misses the last parameter, we felt that correcting for this small effect was not worth the added complexity. A logical source statement in FORTRAN can be computed by counting only those lines that have the blank character in column 1 and either a blank or a zero in column 6. This follows from the convention that comments in FORTRAN are identified as those lines with the character "C" or "*" in column 1, while continuation lines have any character except a blank or a zero in column 6. This rule, therefore, counts only non-comment, non-continuation lines. In structured FORTRAN (such as that used in our examples) the statement "end if" is not counted as a logical source statement but it is included in the count of physical source statements.

The difference between the PSS and LSS methods and how they apply to counting code in Ada and FORTRAN can best be illustrated through a simple example. Table II-1 shows a portion of the FORTRAN and Ada code found in the Fourier analysis subroutine.

Table II-1. Comparison of Ada and FORTRAN "if...then" Statements

FORTRAN	LSS	Ada	LSS
if (j.gt.i)then	✓	if J > I then	
tempr=data(j)	✓	Temp:= Data (J);	✓
tempi=data(j+1)	✓	Data (J):= Data (I);	✓
data(j)=data(i)	✓	Data (I):= Temp;	✓
data(j+1)=data(i+1)	✓	end if;	✓
data(i)=tempr	✓		
data(i+1)=tempi	✓		
end if			

The portion of the subroutine is an "if...then" statement written in the styles according to the references noted above. (Capitalization is not significant in either language. The lower-case convention used in the FORTRAN example is adopted from [4].) The Ada PSS count is five and the FORTRAN PSS count is eight. The Ada LSS count is four and the FORTRAN LSS count is seven. There are four semicolons in the Ada code. The "end if" in the FORTRAN code is not counted as a logical statement since it is required by and part of the "if" statement.

III. RESULTS

In this chapter we examine the results of applying the code and style conventions discussed in the preceding chapter to the four test programs. We examine some of the differences between Ada and FORTRAN that might explain the results, and we discuss the notion that the Ada language exhibits scale economies (i.e., as the size of the program grows the number of Ada lines grows slower than the size of an equivalent FORTRAN program). Finally, we discuss the effect of our results on the practice of software cost estimating.

A. PSS AND LSS COUNTS

The results of applying the code-counting methods to the FORTRAN and Ada examples using the conventional programming style examples are summarized in Table III-1.

Table III-1. Lines-of-Code Count for Four Programs

	FORTRAN		Ada		Ada/FORTRAN	
Program	PSS	LSS	PSS	LSS	PSS	LSS
Quicksort	92	79	141	106	1.53	1.34
Moments	68	61	124	109	1.82	1.78
Fourier	133	115	189	147	1.42	1.27
Orbit	1,101	803	1,382	1,065	1.25	1.32
Mean:					1.51	1.43

There are several interesting aspects to the results. The PSS count is always greater than the LSS count. The Ada code count is in every case greater than the FORTRAN code count. The Ada code count is, on average, 50-percent greater than the FORTRAN count when measured by PSS. The Ada code count is, on average, 40-percent greater than the FORTRAN count when measured by LSS. McGarry and Agresti, in an experiment of parallel development of flight dynamics systems by two teams of programmers, one team using FORTRAN and the other team using Ada, reported the Ada product was significantly larger (measured by PSS) than the FORTRAN product by a factor of almost three [10]. McGarry and Agresti posit three reasons for the large difference in the counts. First, the characteristics of the Ada language itself (about which more will be said in the next

sections). Second, additional functionality was built into the Ada version (the Ada team developed a more contemporary screen-oriented user interface). Third, the Ada version was not driven by tight schedules and funds as was the FORTRAN version; thus, there was a tendency to continually add capability to the Ada version. Our experiment controlled for the latter two factors. Our results also indicate that as the size of the program grows, the difference between the FORTRAN and Ada counts falls. This result suggests that Ada exhibits economies of scale relative to FORTRAN.

In order to determine if the observed differences between the Ada and FORTRAN code counts are statistically significant, we conducted a nonparametric test. We would expect Ada to be greater than FORTRAN half of the time and less than FORTRAN half the time. As noted, the Ada program code counts were always greater than the FORTRAN counts. Obviously, we would reject the null hypothesis that the Ada and FORTRAN counts were the same. One might wonder whether the same results would be observed if the programs were decomposed into their smaller constituencies. We decomposed the four programs into 17 corresponding modules. In only one out the 17 components was the Ada component not larger than the equivalent FORTRAN component. Here again we would reject the null hypothesis that the Ada and FORTRAN counts were the same.[3]

In carrying out this experiment, we observed that FORTRAN, like Ada, has optional variations in style that can change the number of lines in a subroutine depending on the formatting used. We also observed that certain kinds of statements, such as input and output statements, were more verbose in Ada than in FORTRAN, while other kinds of statements, such as assignments to structured data, could be expressed more efficiently in Ada. The affect of these variations in style are discussed in detail in the next two sections.

---

[3] The nonparametric test that we conducted was the sign test. To test the hypothesis that Ada code counts are greater than FORTRAN code counts, we needed to determine whether the null hypothesis (code counts are the same) could be rejected at a specific level of significance α The null hypothesis can be rejected if $x \geq k_\alpha$, where x is the number of positive differences (i.e., Ada code count is greater than FORTRAN code count) and k_α is the smallest integer that satisfies

$$P(x \geq k_\alpha) = \sum_{x=k_\alpha}^{n} \binom{n}{x} \left(\frac{1}{2}\right)^x \left(\frac{1}{2}\right)^{n-x} \leq \alpha,$$

where n is the sample size. When the sign test was performed on the four programs, the null hypothesis could not be rejected at the 5-percent level of significance. However, at the 10-percent level, we rejected the null hypothesis and concluded that the Ada code counts are greater than FORTRAN. We then conducted the sign test on the 17 components and rejected the null hypothesis at the 5-percent level.

B. DECLARATIONS

This section discusses specific issues with respect to how declarations can be counted in both Ada and FORTRAN and why we chose to write and count them as we did.

FORTRAN allows the implicit declaration of variables, where the data type is implied by the first letter of the name (beginning a symbolic name with the letters "I" through "N" implies an integer while any other letter implies a real number). In spite of this allowance, most programming practices now dictate explicit declaration as a way of avoiding certain kinds of errors. Nevertheless, it is common in FORTRAN to use a single statement to declare all the variables of a certain type rather than to place each declaration on a separate line. Conversely, most of the guides about Ada style recommend using a separate line for each declaration. This allows the initialization of variables during the elaboration of their declarations, and also improves the maintainability of the code, though it tends to inflate both the LSS and the PSS for Ada when compared with FORTRAN. As discussed earlier, in order to understand the variability in program size due to the observance of these and other conventions, we wrote and compared both terse and verbose versions of each routine in each language.

Another stylistic issue that tends to increase the size of a program written in Ada over a similar one written in FORTRAN is the use of descriptive names. Since FORTRAN symbolic names are limited in length to six characters [7, section 2.2] it is often easier to fit a long expression that contains several names on a single line. In several of our examples, multiple editor lines were required to write an expression in Ada that took only one line in FORTRAN. One might argue that the descriptive choice of names in Ada might reduce the need for in-line commentary as compared with a corresponding FORTRAN program, meaning that the effect on the size of a fully commented program may be counterbalanced. However, since we did not study the effects of commenting on program size, we did not attempt to investigate this possibility. Further, this issue only affects the physical source statements (PSS) and not the count of statements and declarations (LSS).

In Ada, formal parameters are declared along with the name of a subprogram, rather than in a subsequent declarative area, as is the case with FORTRAN. The effect of this on size is often canceled out, depending on the counting method used, because this practice makes the program unit declaration longer in Ada, but it eliminates the need to repeat the parameter names in a later declaration.

Ada allows, but does not require, the declaration of a library-level subprogram (i.e., a procedure or function) to be compiled separately from its executable body. If this

separation is not done, the number of lines needed to write a given program will be reduced. However, most style guides recommend the addition of these lines because it can greatly reduce the recompilation effort required if a subprogram body is modified. A separate subprogram declaration can also be used within a declarative area, usually to allow mutual visibility between two locally-declared subprograms. In the case of subprograms exported from a package declaration, the subprogram declarations are always separated from their bodies, which appear in the package body. In all these cases, however, the extra programming effort required to provide a separate subprogram declaration is negligible because it is simply a verbatim repetition of the specification part of the subprogram body. In fact, some Ada development environments automatically complete the repeated syntax so that no additional typing or editing is required of the programmer. (It might be argued that maintenance is made more complicated by this syntactic duplication in the language since both copies have to be modified in the case of a change. However, the more likely maintenance situation is a change to the unique code in a subprogram body rather than the redundant interface code in the specification.) For these reasons, one might argue that separate subprogram declarations should not be included in the size of a program. However, we deemed it not worth the added complexity of defining counting rules to compensate for this.

Since FORTRAN does not allow the definition of structured data types, arrays are often used for various logical data structures. This simplifies the declaration of such structures, because it requires only a dimension statement, but at the possible expense of more elaborate processing later in the program. To assign an array value to an array object in FORTRAN, it is necessary to use a loop that explicitly assigns each component. In comparison, Ada array objects contain implicit information about their own size and bounds, which allows the array objects to be assigned to one another with single assignment statements.

The manipulation of arrays that represent nested data structures can require even more complexity. For example, the FORTRAN version of the fast Fourier transform used in one of our examples uses an array of real numbers to represent an array of complex numbers. The odd-indexed values are the real parts and the even-indexed values are the imaginary parts. This practice required the "do" loops to use an increment of two rather than one each time the complex numbers were processed. When we initially translated the programs into Ada, extra statements were required to implement these loops because Ada does not allow "for" loops that skip values in the loop range. When the algorithms were

written in a more appropriate Ada style using structured data, however, the loops were reduced to half the number of statements required by their FORTRAN equivalents.

C. STATEMENTS

This section presents issues with respect to how statements can be written and counted in both Ada and FORTRAN.

One of the most noticeable differences in program size between Ada and FORTRAN programs was in the statement areas used for input and output. Because Ada allows only a single value to be either input or output with each statement, the translation into Ada of a formatted FORTRAN input/output statement often resulted in considerable expansion. This effect can be clearly observed by comparing the driver routines in Ada and FORTRAN for the example engineering algorithms. For example, the three "write" statements in the driver for the Fourier transform routine required five statements, while the same output in Ada required 27 statements.

One of the stylistic issues that allows a single FORTRAN program to be written with different numbers of lines is the use of separate "format" statements when specifying input and output columns rather than including this information directly in the "read" or "write" statements. Our FORTRAN program examples, which were originally written without "format" statements, were re-styled to conform to the conventions found in [7] in order to make them representative of industry programming standards. Note that the verbose and terse versions of the FORTRAN programs show the different ways to effect input and output formatting.

We noticed several minor differences between the syntactic conventions used in the two languages when applying the counting rules chosen. One minor difference between the languages is that Ada always implicitly declares loop variables. This is to ensure that the availability of that variable is limited to the scope of its loop. It also has the effect of reducing the size of the program by one declaration. Another minor difference is the implicit "return" statement at the end of an Ada subprogram. A "return" statement is still required if processing is to stop at any other point, but most Ada subprograms are written to return after their last statement. In a FORTRAN routine, the last line must be an "end" statement. It has the same effect as a "return" statement, which is to return control to the referencing program unit. Nevertheless, it is common to see both a "return" and an "end" statement in a FORTRAN subprogram. A third minor difference is the lack of a need for a "continue" statement in Ada. Although a "continue" statement is rarely required in FORTRAN, it is common practice to use one at the end of a loop to avoid confusing the

last statement in the loop with the statements that follow the loop. In comparison, Ada loops require an "end loop"; however, this increases the number of physical lines (PSS) only, not the number of statements (LSS).

It should be noted that an inconsistency between the LSS methods for Ada and FORTRAN existed. In FORTRAN the "else" statement is counted as a logical source statement, but it is considered part of the same logical statement as its containing "if" statement in Ada. Thus, it was not counted as an additional logical source statement in Ada [8].

D. ECONOMIES OF SCALE IN THE ADA LANGUAGE

As previously noted, the relative difference between the FORTRAN and Ada counts fell as the size of the four programs in the experiment grew. An interesting question concerning this observed scale effect is: At what program size would the Ada code count fall below the FORTRAN code count? A graphical representation of this cross-over or break-even point is presented in Figure III-1.

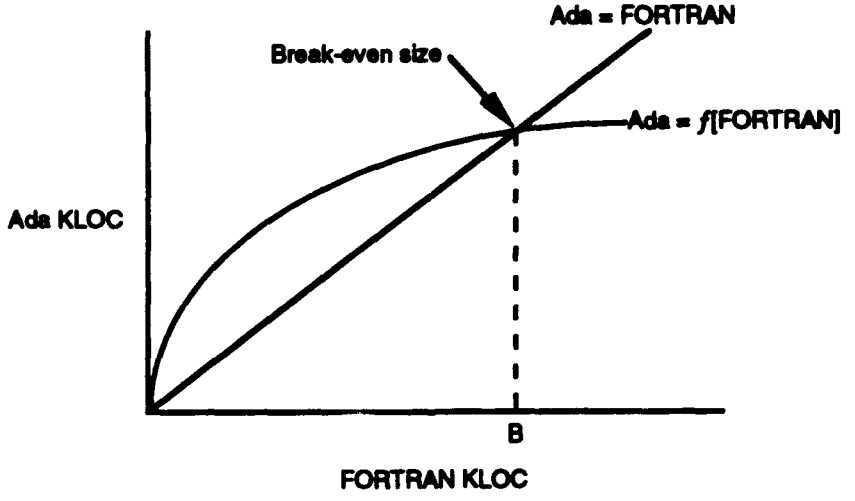


Figure III-1. Break-Even Size

The Ada and FORTRAN lines of code (measured in thousands) are represented on the Y and the X axis, respectively. The ray that passes through the origin at 45 degrees represents points where the number of Ada and FORTRAN lines of code are equal. The curved line represents a hypothetical relationship between Ada and FORTRAN. If economies of scale exist, we would expect this relationship to exhibit a curvilinear form similar to that depicted in Figure III-1. This form suggests that as the size of the program grows, the number of Ada lines required to perform the function grows, more slowly than

does the corresponding FORTRAN program. The point where the two lines intersect represents the break-even point. The interpretation of this point is that any program of size greater than B could be written with fewer lines in Ada than in FORTRAN.

What is the actual shape of the curvilinear line? We estimated a regression curve of the form:

$$KLOC_A = \alpha KLOC_F^{\beta} \mu,$$

where α and β are the coefficients to be estimated, and μ is a stochastic disturbance term.

The results of estimating the coefficients using the four test programs were:

$$KLOC_A = 1.26 KLOC_F^{.937} \qquad (5)$$

$$(1.27) \qquad (11.8)$$

$$\bar{R} = .97 \quad SEE = .16$$

The numbers in parentheses are the t-scores of the FORTRAN coefficient and intercept term. If there were a scale effect, we would expect the coefficient on the FORTRAN variable to be less than one. In this case, the value of coefficient is 0.937. Using these regression results the break-even point was computed to be around 40,000 lines of code.[4]

Note that we have estimated the point where functionally equivalent programs in Ada and FORTRAN would be the same size. However, this is the break-even size only from the perspective of development effort if the cost to develop a line of code in either language is the same. To compute the break-even cost, we must drop the implicit assumption that the cost to develop a line of code is the same for both languages. The cross-over point would depend on the values of the parameters used in the FORTRAN and Ada cost or effort estimating equations.[5]

---

4 From the regression we found:

$$KLOC_A = 1.26 KLOC_F^{.937}.$$

If there were no difference between Ada and FORTRAN, then the relationship would simply be $A = F$. Setting these two relationships equal to each other, we have

$$1.26 KLOC_F^{.937} = KLOC_F ,$$

and solving for the break-even point, we find:

$$KLOC_F = \sqrt[\frac{1}{.063}]{1.26}$$

$$= 39{,}190.$$

5 As an example, assume that the effort estimating equation for a FORTRAN development is taken to be

$$E_F = 3.0 KLOC_F^{1.12},$$

A word of caution about the interpretation of these results is in order. The results are based on only four relatively small programs, all of which are smaller than the projected break-even point. However, there is at least some anecdotal evidence from NASA/SEL and others that suggests that the magnitude of our estimate is consistent with their findings in this area [5].

The question remains as to why we should observe this scale effect with the Ada language. Several especially noticeable differences between the two languages may contribute to the effect. One difference was the fact that as program size increased, the executable portions increased slower in Ada than in FORTRAN. Although the declarative portions increased more in Ada than in FORTRAN, they contributed less to overall size. In our largest example, the executable portion was smaller in Ada than in FORTRAN even though the overall size was greater in Ada. Table III-2 presents a view of the four test programs separated into their executable and declarative portions.

Table III-2. Executable and Declaration Code Count

	FORTRAN LOC		Ada LOC		Ada/FORTRAN	
Program	Executable	Declarative	Executable	Declarative	Executable	Declarative
Quicksort	72	7	81	25	1.12	3.57
FFT	104	11	110	37	1.05	3.36
Moment	52	9	71	38	1.36	4.22
Orbit	738	65	701	364	0.95	5.66

An example from the FFT test program illustrates this tradeoff between the number of executable and declarative statements. Table III-3 presents functionally equivalent Ada and FORTRAN code taken from the FFT program.

If our results that indicate significant differences in size between functionally equivalent FORTRAN and Ada programs are correct, then the practice of cost analysts to

---

and the effort estimating equation for an Ada development is assumed to be

$$E_A = 5.8 KLOC_A^{1.04}.$$

Setting these two equations equal to each other and substituting in our estimated relationship between Ada and FORTRAN, we get:

$$5.8[1.26 KLOC_F^{.937}]^{1.04} = 3.0 KLOC_F^{1.12}.$$

Solving for F, we find the point of equal effort for Ada and FORTRAN developments is about 485,000 lines of code.

use Ada and FORTRAN lines-of-code counts interchangeably will induce errors in the subsequent cost estimates.

Table III-3. Declarations Versus Statements

FORTRAN	Ada
REAL Data (2*nn) REAL tempi, tempr REAL wi, wr	type Complex is record Real : Float; Imaginary : Float; end record; type Complex_Array is array (Natural range<>)of Complex; function "+" (Left,Right:Complex) return Complex; function "-" (Left,Right:Complex) return Complex; function "*" (Left,Right:Complex) return Complex; Data : Complex_Array (1..N); W, Temp : Complex;
.	.
.	.
.	.
tempr=wr*Data(j) - wi*Data(j+1)	Temp :=W*Data (J);
tempr=wr*Data(j+1) - wi*Data(j)	Data (J) := Data (I) - Temp;
Data(j)=Data(i) - tempr	Data (I) := Data (I) - Temp;
Data(j+1)=Data(i+1) - tempi	
Data(i)=Data(i) + tempr	
Data(i+1)=Data(i+1) + tempi	

How large these potential errors can be is seen in Table III-4. The table's first column is the number of lines of FORTRAN code. The second column is the estimate of the effort (measured in staff-months) to develop the appropriate FORTRAN lines of code using an equation taken from [1]. The third column is the estimate of the effort required to develop Ada code using the FORTRAN code count as the explanatory variable value rather than the appropriate Ada code count value. In this case, the parameter values were taken from [2]. The fourth column shows our algorithm that converts FORTRAN lines to the equivalent Ada lines of code count. The last column shows the effort-estimates that result from using the Ada code count from column four in the Ada effort-estimating equation.

Table III-4. Estimates of Error

LOC in FORTRAN	Estimated Effort in FORTRAN Using $E_F = 3.0(KLOC_F)^{1.12}$	Estimated Effort in Ada Using $E_A = 5.8(KLOC_F)^{1.04}$	Estimated LOC in Ada Using $KLOC_A = 1.26KLOC^{.94}$	Estimated Effort in Ada Using $E_A = 5.8(KLOC_A)^{1.04}$
1,000	3.0	5.8	1,260	7.4
5,000	18.2	30.9	5,720	35.6
10,000	39.5	63.6	10,974	70.1
20,000	86.0	130.8	21,054	137.9
40,000	186.8	268.9	40,393	271.6
100,000	521.3	697.3	95,581	665.3
500,000	3,162.0	3,718.4	425,898	3,147.1

The difference between the effort estimates in column three (which simulate the practice of using Ada and FORTRAN lines of code counts interchangeably) and those in the last column (which represent the "correct" estimate) is the error. We note that for small programs the relative error is large (e.g., 27 percent for 1,000 lines of code), then gets smaller as it approaches the cross-over point at around 40,000 lines, then grows larger again, but at a very slow rate (at 100,000 lines, the error is approximately 5 percent). Obviously, the results are sensitive to the effort-estimating equation used. Again, our FORTRAN-to-Ada equation is based on a small sample and should be used with caution. However, the point is that significant error can result from the practice of indiscriminately interchanging code-counting units.

IV. CONCLUSIONS AND FUTURE RESEARCH

The main objective of this research was to fill a gap in the knowledge needed by experienced FORTRAN size and cost estimators when estimating Ada developments for the first time. Although there are published models for the cost of developing Ada programs based on their expected size, there has been no standard way of estimating the size of an Ada development based on the expected size of an equivalent FORTRAN development. This work has shown that the sizes of functionally equivalent programs in Ada and FORTRAN are different. It would therefore be a mistake to assume that either a FORTRAN effort estimate or the expected number of FORTRAN lines of code would be sufficient in an Ada estimating equation. One such Ada equation was shown in this study; however, the magnitude of the error will depend on the exact estimating equations used. With the added knowledge of how the sizes of functionally equivalent programs in Ada and FORTRAN compare, a cost estimator can first adjust the expected number of lines of FORTRAN code to complete a job to a more accurate estimate of the expected number of lines of Ada code. Then, an Ada effort-estimating equation may be properly applied.

This study should be viewed mainly as a model for further investigation, although we believe our limited results are still of interest. In particular, we suspect that the tendency we observed for small Ada programs to be larger than their functionally equivalent FORTRAN counterparts is reasonable, as is our further observation that the overhead for Ada diminishes as the program size is increased. Our limited data suggested that there may even be a cross-over point beyond which the size of an Ada program is smaller than a functionally equivalent FORTRAN program. Although the number of observations was small and all were below this projected cross-over point, one of the strongest pieces of evidence that such a point exists, beyond the regression analysis, is that the number of executable lines of Ada in our largest example was smaller than the equivalent number of executable lines in FORTRAN. After an inspection of language features, we believed this to be a reasonable occurrence, because Ada has richer declarative power and, in return, can take advantage of simpler algorithmic processing.

Because the relationship between the sizes of functionally equivalent Ada and FORTRAN programs is probably not linear, more observations are needed, and in particular, observations are needed that are at least an order of magnitude greater than the

largest of our examples. The only published comparison of the sizes of a pair of large Ada and FORTRAN programs developed from the same set of requirements was too confounded to be useful for this purpose [10]. The only other evidence we found about the way larger Ada and FORTRAN programs compare was anecdotal, although the opinions reported to us tended to agree with our observations [5].

A final observation not previously mentioned stems from our interest in examining the possible variations of program size due to programming style. Although we used what we considered to be a conventional style of formatting for the programs in our analyses, we additionally wrote both terse and verbose styles for each example. The most interesting result we observed was that the possible variation in size for an Ada program is much greater than the possible variation for a FORTRAN program. This means that the comparison of Ada size, effort, and productivity results across organizations (which may not be observing the same style standards) is more prone to error than are similar comparisons using FORTRAN results. Although we used well-defined counting rules for both languages to maximize the portability of our results, we were not able to similarly well-define a programming style. In order to assure comparability of Ada size, effort, and productivity results across organizations, more study is needed into how the size of an Ada program might be normalized for any implemented functionality.

APPENDIX A

ADA AND FORTRAN CODE FOR EXAMPLE PROGRAMS

APPENDIX A

ADA AND FORTRAN CODE FOR EXAMPLE PROGRAMS

This appendix contains the code for three of the four programs dicussed in the report. The three programs are Quicksort, Fast Fourier Transform, and Moments of a Distribution. The code for the Orbit program was not included becuase of its length. A copy of the code is available from the authors.

QUICKSORT

FORTRAN

```
      PROGRAM xsort
C     driver for routine sort
      INTEGER i,j
      REAL a(100)
       open(7,file='TARRAY1.DAT',status='OLD')
        open(8,file='sortprog.out',status='NEW')
       read(7,*) (a(i),i=1,100)
      close(7)
C     print original array
       write(8,*) 'Original array:'
       do 11 i=1,10
         write(8,15) (a(10*(i-1)+j),j=1,10)
11    continue
15    format(1x,10f7.2)
C     sort array
      call sort(100,a)
C     print sorted array
       write(8,*) 'Sorted array:'
       do 12 i=1,10
         write(8,15) (a(10*(i-1)+j),j=1,10)
12    continue
      close(8)
      END

      SUBROUTINE sort(n,arr)
      INTEGER n,M,NSTACK
      REAL arr(n)
      PARAMETER (M=7,NSTACK=50)
      INTEGER i,ir,j,jstack,k,l,istack(NSTACK)
      REAL a,temp
```

```
      jstack=0
      l=1
      ir=n
1     if(ir-l.lt.M)then
        do 12 j=l+1,ir
          a=arr(j)
          do 11 i=j-1,1,-1
            if(arr(i).le.a)goto 2
            arr(i+1)=arr(i)
11        continue
          i=0
2         arr(i+1)=a
12      continue
        if(jstack.eq.0)return
        ir=istack(jstack)
        l=istack(jstack-1)
        jstack=jstack-2
      else
        k=(l+ir)/2
        temp=arr(k)
        arr(k)=arr(l+1)
        arr(l+1)=temp
        if(arr(l+1).gt.arr(ir))then
          temp=arr(l+1)
          arr(l+1)=arr(ir)
          arr(ir)=temp
        endif
        if(arr(l).gt.arr(ir))then
          temp=arr(l)
          arr(l)=arr(ir)
          arr(ir)=temp
        endif
        if(arr(l+1).gt.arr(l))then
          temp=arr(l+1)
          arr(l+1)=arr(l)
          arr(l)=temp
        endif
        i=l+1
        j=ir
        a=arr(l)
3       continue
          i=i+1
        if(arr(i).lt.a)goto 3
4       continue
          j=j-1
        if(arr(j).gt.a)goto 4
        if(j.lt.i)goto 5
        temp=arr(i)
        arr(i)=arr(j)
        arr(j)=temp
        goto 3
5       arr(l)=arr(j)
        arr(j)=a
        jstack=jstack+2
```

```
        if(jstack.gt.NSTACK)pause 'NSTACK too small in sort'
        if(ir-i+1.ge.j-l)then
         istack(jstack)=ir
         istack(jstack-1)=i
         ir=j-1
        else
         istack(jstack)=j-1
         istack(jstack-1)=l
         l=i
        endif
       endif
       goto 1
       END
```

ADA

Quicksort Routine

```
with Text_Io;
procedure Quicksort_Generic (Arr : in out Element_Array) is
 Temp_Stack : array (1 .. Max_Size) of Integer;
 Stack_Ptr : Integer := 0;
 L  : Integer := 1;
 IR : Integer := Arr'Length;
 A  : Element;
 I  : Integer;
 J  : Integer;
 IQ : Integer;
 Found : Boolean;

 function "<" (Left, Right : Element) return Boolean is
 begin
  if Left = Right then
   return False;
  end if;
  return Left <= Right;
 end "<";

 function ">" (Left, Right : Element) return Boolean is
 begin
  return not "<=" (Left, Right);
 end ">";

begin
 loop
  if IR - L < Subarray_Size then
   for JJ in L + 1 .. IR loop
    A := Arr (JJ);
    Found := False;
    for II in reverse 1 .. JJ - 1 loop
     if Arr (II) <= A then
      Found := True;
      Arr (II + 1) := A;
      exit;
```

```
      else
        Arr (II + 1) := Arr (II);
      end if;
    end loop;
    if not Found then
      Arr (1) := A;
    end if;
  end loop;
  if Stack_Ptr = 0 then
    return;
  end if;
  IR := Temp_Stack (Stack_Ptr);
  L := Temp_Stack (Stack_Ptr - 1);
  Stack_Ptr := Stack_Ptr - 2;
else
  I := L;
  J := IR;
  IQ := (L + IR) / 2;
  A := Arr (IQ);
  Arr (IQ) := Arr (L);
  loop
    while J > 0 loop
      if A < Arr (J) then
        J := J - 1;
      else
        exit;
      end if;
    end loop;
    if J <= I then
      Arr (I) := A;
      exit;
    end if;
    Arr (I) := Arr (J);
    I := I + 1;
    while I <= Arr'Length loop
      if A > Arr (I) then
        I := I + 1;
      else
        exit;
      end if;
    end loop;
    if J <= I then
      Arr (J) := A;
      I := J;
      exit;
    end if;
    Arr (J) := Arr (I);
    J := J - 1;
  end loop;
  Stack_Ptr := Stack_Ptr + 2;
  if Stack_Ptr > Max_Size then
    Text_Io.Put_Line ("Max_Size must be made larger.");
    raise Constraint_Error;
  end if;
```

```
    if IR - I >= I - L then
     Temp_Stack (Stack_Ptr) := IR;
     Temp_Stack (Stack_Ptr - 1) := I + 1;
     IR := I - 1;
    else
     Temp_Stack (Stack_Ptr) := I - 1;
     Temp_Stack (Stack_Ptr - 1) := L;
     L := I + 1;
    end if;
   end if;
  end loop;
end Quicksort_Generic;
```

Specification for Quicksort

```
generic
 type Element is private;
 type Element_Array is array (Positive range <>) of Element;
 with function "<=" (Left, Right : Element) return Boolean;
 Max_Size : Natural := 50;
 Subarray_Size : Positive := 7;
procedure Quicksort_Generic (Arr : in out Element_Array);
```

Driver for Quicksort

```
with Quicksort_Generic;
with Text_Io;
procedure Xquicksort_Generic is
 package Float_Io is new Text_Io.Float_Io (Float);
 Size : constant := 100;
 type Float_Array is array (Positive range <>) of Float;
 procedure Sort is new Quicksort_Generic (Float, Float_Array, "<=");
 A : Float_Array (1 .. Size);
 File : Text_Io.File_Type;
 Output : Text_Io.File_Type;
begin
 Text_Io.Open (File, Text_Io.In_File, "Tarray.Dat");
 for I in 1 .. Size loop
  Float_Io.Get (File, A (I));
 end loop;
 Text_Io.Close (File);
--print original array
 Text_Io.Create (Output, Text_Io.Out_File, "Output.Lis");
 Text_Io.Set_Output (Output);
 Text_Io.Put_Line ("Original array:");
 for I in 1 .. 10 loop
  for J in 1 .. 10 loop
   Float_Io.Put (A (10 * (I-1) + J), 4, 2, 0);
  end loop;
  Text_Io.New_Line;
 end loop;
--sort array
 Sort (A);
--print sorted array
 Text_Io.Put_Line ("Sorted array:");
 for I in 1 .. 10 loop
```

```
      for J in 1 .. 10 loop
        Float_Io.Put (A (10 * (I-1) + J), 4, 2, 0);
      end loop;
      Text_Io.New_Line;
    end loop;
    Text_Io.Close (Output);
  end Xquicksort_Generic;
```

MOMENTS OF A DISTRIBUTION

FORTRAN

```
      PROGRAM xmoment
C     driver for routine moment
      REAL PI
      INTEGER Nbin,Ndat,Npts
      PARAMETER(PI=3.14159265,Npts=10000,Nbin=100,Ndat=Npts+Nbin)
      INTEGER i,j,k,nlim
      REAL adev,ave,curt,data(Ndat),sdev,skew,var,x
      i=1
      do 12 j=1,Nbin
        x=PI*j/Nbin
        nlim=nint(sin(x)*PI/2.0*Npts/Nbin)
        do 11 k=1,nlim
          data(i)=x
          i=i+1
11      continue
12    continue
      open(9,file='statsprog.out',status='NEW')
      write(9,15) 'Moments of a sinusoidal distribution'
      call moment(data,i-1,ave,adev,sdev,var,skew,curt)
      write(9,16) 'Calculated','Expected'
      write(9,17) 'Mean :',ave,PI/2.0
      write(9,17) 'Average Deviation :',adev,0.570796
      write(9,17) 'Standard Deviation :',sdev,0.683667
      write(9,17) 'Variance :',var,0.467401
      write(9,17) 'Skewness :',skew,0.0
      write(9,17) 'Kurtosis :',curt,-0.806249
15    format(1x,a/)
16    format(1x,t29,a,t42,a/)
17    format(1x,a,t25,2f15.4)
      close(9)
      END

      SUBROUTINE moment(data,n,ave,adev,sdev,var,skew,curt)
      INTEGER n
      REAL adev,ave,curt,sdev,skew,var,data(n)
      INTEGER j
      REAL p,s,ep
      if(n.le.1)pause 'n must be at least 2 in moment'
      s=0.
      do 11 j=1,n
```

```
      s=s+data(j)
   11 continue
      ave=s/n
      adev=0.
      var=0.
      skew=0.
      curt=0.
      ep=0.
      do 12 j=1,n
       s=data(j)-ave
       ep=ep+s
       adev=adev+abs(s)
       p=s*s
       var=var+p
       p=p*s
       skew=skew+p
       p=p*s
       curt=curt+p
   12 continue
      adev=adev/n
      var=(var-ep**2/n)/(n-1)
      sdev=sqrt(var)
      if(var.ne.0.)then
       skew=skew/(n*sdev**3)
       curt=curt/(n*var**2)-3.
      else
       pause 'no skew or kurtosis when zero variance in moment'
      endif
      return
      END
```

ADA

Moment Subroutine

```
with Text_Io;
procedure Moment_Generic (Data : Data_Array;
                          Ave  : in out Real;
                          Adev : in out Real;
                          Sdev : in out Real;
                          Var  : in out Real;
                          Skew : in out Real;
                          Curt : in out Real) is
 Powers   : Real;
 Sum      : Real;
 Deviation : Real;
 Sum_Devs  : Real := 0.0;
 N        : constant Integer := Data'Length;
begin
 if N <= 1 then
  Text_Io.Put_Line ("Must be at least 2 in moment.");
 else
  Sum := 0.0;
  for J in Data'Range loop
   Sum := Sum + Data (J);
```

```
   end loop;
   Ave    := Sum / Real (N);
   Adev   := 0.0;
   Var    := 0.0;
   Skew   := 0.0;
   Curt   := 0.0;
   for J in Data'Range loop
    Deviation := Data (J) - Ave;
    Sum_Devs  := Sum_Devs + Deviation;
    Adev      := Adev + abs (Deviation);
    Powers    := Deviation * Deviation;
    Var       := Var + Powers;
    Powers    := Powers * Deviation;
    Skew      := Skew + Powers;
    Powers    := Powers * Deviation;
    Curt      := Curt + Powers;
   end loop;
   Adev := Adev / Real (N);
   Var  := (Var - Sum_Devs ** 2 / Real (N)) / Real (N - 1);
   Sdev := Sqrt (Var);
   if Var /= 0.0 then
    Skew := Skew / (Real (N) * Sdev * Sdev * Sdev);
    Curt := Curt / (Real (N) * Var ** 2) - 3.0;
   else
    Text_Io.Put_Line ("No skew or kurtosis when zero variance in moment");
   end if;
  end if;
end Moment_Generic;
```

Procedure Specification for Moment Routine

```
generic
 type Real is digits <>;
 type Data_Array is array (Positive range <>) of Real;
 with function Sqrt (Number : Real) return Real;
procedure Moment_Generic (Data : Data_Array;
                          Ave  : in out Real;
                          Adev : in out Real;
                          Sdev : in out Real;
                          Var  : in out Real;
                          Skew : in out Real;
                          Curt : in out Real);
```

Driver for Moment Routine

```
with Text_Io;
with Moment_Generic;
with Math_Lib;
procedure Xmoment_Generic is
 Ave, Adev, Sdev, Var, Skew, Curt, X : Float;
 Pi : constant Float := 3.14159265;
 Nbin : constant Natural := 100;
 Npts : constant Natural := 10000;
 Ndat : constant Natural := Npts + Nbin;
 Nlim : Integer;
 I : Integer := 1;
```

```
  type Float_Array is array (Positive range <>) of Float;
  Data : Float_Array (1 .. Ndat);
  package Int_Io is new Text_Io.Integer_Io (Integer);
  package Float_Io is new Text_Io.Float_Io (Float);
  package Math is new Math_Lib (Float);
  procedure Moment is new Moment_Generic (Float,
                                          Float_Array,
                                          Math.Sqrt);
begin
  for J in 1 .. Nbin loop
    X := Pi * Float (J) / Float (Nbin);
    Nlim := Integer (Math.Sin(X) * Pi/2.0 * Float (Npts/Nbin));
    for K in 1 .. Nlim loop
      Data (I) := X;
      I := I + 1;
    end loop;
  end loop;
  Text_Io.Put_Line ("Moments of a sinusoidal distribution");
  Moment (Data (1 .. I-1), Ave, Adev, Sdev, Var, Skew, Curt);
  Text_Io.Set_Col (29);
  Text_Io.Put ("Calculated");
  Text_Io.Set_Col (42);
  Text_Io.Put_Line ("Expected");
  Text_Io.Put ("Mean :");
  Text_Io.Set_Col (25);
  Float_Io.Put (Ave, 6, 5, 0);
  Float_Io.Put (Pi / 2.0, 7, 5, 0);
  Text_Io.New_Line;
  Text_Io.Put ("Average Deviation :");
  Text_Io.Set_Col (25);
  Float_Io.Put (Adev, 6, 5, 0);
  Float_Io.Put (0.570796, 7, 5, 0);
  Text_Io.New_Line;
  Text_Io.Put ("Standard Deviation: ");
  Text_Io.Set_Col (25);
  Float_Io.Put (Sdev, 6, 5, 0);
  Float_Io.Put (0.683667, 7, 5, 0);
  Text_Io.New_Line;
  Text_Io.Put ("Variance :");
  Text_Io.Set_Col (25);
  Float_Io.Put (Var, 6, 5, 0);
  Float_Io.Put (0.467401, 7, 5, 0);
  Text_Io.New_Line;
  Text_Io.Put ("Skewness: ");
  Text_Io.Set_Col (25);
  Float_Io.Put (Skew, 6, 5, 0);
  Float_Io.Put (0.0, 7, 5, 0);
  Text_Io.New_Line;
  Text_Io.Put ("Kurtosis :");
  Text_Io.Set_Col (25);
  Float_Io.Put (Curt, 6, 5, 0);
  Float_Io.Put (-0.806249, 7, 5, 0);
  Text_Io.New_Line;
end Xmoment_Generic;
```

FAST FOURIER TRANSFORM

FORTRAN

```
      PROGRAM xfour1
C     driver for routine four1
      INTEGER NN,NN2
      PARAMETER (NN=32,NN2=2*NN)
      REAL data(NN2),dcmp(NN2)
      INTEGER i,isign,j
       open(8,file='four1prog.out',status='NEW')
      write(8,*) 'h(t)=real-valued even-function'
      write(8,*) 'H(n)=H(N-n) and real?'
      do 11 i=1,2*NN-1,2
        data(i)=1.0/(((i-NN-1.0)/NN)**2+1.0)
        data(i+1)=0.0
11    continue
      isign=1
      call four1(data,NN,isign)
      call prntft(data,NN2)
      write(8,*) 'h(t)=imaginary-valued even-function'
      write(8,*) 'H(n)=H(N-n) and imaginary?'
      do 12 i=1,2*NN-1,2
        data(i+1)=1.0/(((i-NN-1.0)/NN)**2+1.0)
        data(i)=0.0
12    continue
      isign=1
      call four1(data,NN,isign)
      call prntft(data,NN2)
      write(8,*) 'h(t)=real-valued odd-function'
      write(8,*) 'H(n)=-H(N-n) and imaginary?'
      do 13 i=1,2*NN-1,2
        data(i)=(i-NN-1.0)/NN/(((i-NN-1.0)/NN)**2+1.0)
        data(i+1)=0.0
13    continue
      data(1)=0.0
      isign=1
      call four1(data,NN,isign)
      call prntft(data,NN2)
      write(8,*) 'h(t)=imaginary-valued odd-function'
      write(8,*) 'H(n)=-H(N-n) and real?'
      do 14 i=1,2*NN-1,2
        data(i+1)=(i-NN-1.0)/NN/(((i-NN-1.0)/NN)**2+1.0)
        data(i)=0.0
14    continue
      data(2)=0.0
      isign=1
      call four1(data,NN,isign)
      call prntft(data,NN2)
C     transform, inverse-transform test
      do 15 i=1,2*NN-1,2
        data(i)=1.0/((0.5*(i-NN-1)/NN)**2+1.0)
        dcmp(i)=data(i)
```

```
      data(i+1)=(0.25*(i-NN-1)/NN)*
     *     exp(-(0.5*(i-NN-1.0)/NN)**2)
      dcmp(i+1)=data(i+1)
15  continue
    isign=1
    call four1(data,NN,isign)
    isign=-1
    call four1(data,NN,isign)
    write(8,20) 'Original Data:','Double Fourier Transform:'
20  format(/1x,t10,a,t44,a)
     write(8,21) 'k','Real h(k)','Imag h(k)','Real h(k)','Imag h(k)'
21  format(/1x,t5,a,t11,a,t24,a,t41,a,t53,a/)
    do 16 i=1,NN,2
     j=(i+1)/2
     write(8,22) j,dcmp(i),dcmp(i+1),data(i)/NN,data(i+1)/NN
22    format(1x,i4,2x,2f12.6,5x,2f12.6)
16  continue
    close(8)
    END
    SUBROUTINE prntft(data,nn2)
    INTEGER n,nn2,m,mm
    REAL data(nn2)
     write(8,30) 'n','Real H(n)','Imag H(n)','Real H(N-n)',
     *     'Imag H(N-n)'
30  format(/1x,t5,a,t11,a,t23,a,t39,a,t52,a)
    write(8,31) 0,data(1),data(2),data(1),data(2)
31  format(1x,i4,2x,2f12.6,5x,2f12.6)
    do 11 n=3,(nn2/2)+1,2
     m=(n-1)/2
     mm=nn2+2-n
     write(8,31) m,data(n),data(n+1),data(mm),data(mm+1)
11  continue
    return
    END

    SUBROUTINE four1(data,nn,isign)
    INTEGER isign,nn
    REAL data(2*nn)
    INTEGER i,istep,j,m,mmax,n
    REAL tempi,tempr
    DOUBLE PRECISION theta,wi,wpi,wpr,wr,wtemp
    n=2*nn
    j=1
    do 11 i=1,n,2
     if(j.gt.i)then
      tempr=data(j)
      tempi=data(j+1)
      data(j)=data(i)
      data(j+1)=data(i+1)
      data(i)=tempr
      data(i+1)=tempi
     endif
     m=n/2
```

```
1     if ((m.ge.2).and.(j.gt.m)) then
       j=j-m
       m=m/2
      goto 1
      endif
      j=j+m
11  continue
     mmax=2
2    if (n.gt.mmax) then
      istep=2*mmax
      theta=6.28318530717959d0/(isign*mmax)
      wpr=-2.d0*sin(0.5d0*theta)**2
      wpi=sin(theta)
      wr=1.d0
      wi=0.d0
      do 13 m=1,mmax,2
       do 12 i=m,n,istep
        j=i+mmax
        tempr=sngl(wr)*data(j)-sngl(wi)*data(j+1)
        tempi=sngl(wr)*data(j+1)+sngl(wi)*data(j)
        data(j)=data(i)-tempr
        data(j+1)=data(i+1)-tempi
        data(i)=data(i)+tempr
        data(i+1)=data(i+1)+tempi
12     continue
       wtemp=wr
       wr=wr*wpr-wi*wpi+wr
       wi=wi*wpr+wtemp*wpi+wi
13   continue
     mmax=istep
    goto 2
    endif
    return
    END
```

ADA

Fast Fourier Routine

```
procedure Fourier_Generic (Data     : in out Complex_Array;
                           Positive : in     Boolean := True) is
 Istep : Integer;
 J : Integer := 1;
 I : Integer;
 M : Integer;
 Mmax : Integer;
 Temp, W, Wp : Complex_Type;
 Theta : Real;
 Pi : constant := 3.1415926535897932;
 Sign : Integer := 1;
begin
 if not Positive then
   Sign := -1;
 end if;
 for I in Data'Range loop
```

```
    if J > I then
      Temp := Data (J);
      Data (J) := Data (I);
      Data (I) := Temp;
    end if;
    M := Data'Length / 2;
    while M >= 1 and J > M loop
      J := J - M;
      M := M / 2;
    end loop;
    J := J + M;
  end loop;
  Mmax := 1;
  while Data'Length > Mmax loop
    Istep := 2 * Mmax;
    Theta := Pi / Real (Sign * Mmax);
    Wp := Make (-2.0 * (Sin (0.5 * Theta)) ** 2, Sin (Theta));
    W := Make (1.0, 0.0);
    for M in 1 .. Mmax loop
      for Iteration in 0 .. (Data'Length - M) / Istep loop
        I := Iteration * Istep + M;
        J := I + Mmax;
        Temp := W * Data (J);
        Data (J) := Data (I) - Temp;
        Data (I) := Data (I) + Temp;
      end loop;
      W := W * Wp + W;
    end loop;
    Mmax := Istep;
  end loop;
end Fourier_Generic;
```

Specification for Fast Fourier Routine

```
generic
  type Real is digits <>;
  with function Sin (Angle : Real) return Real;
  type Complex_Type is private;
  type Complex_Array is array (Positive range <>) of Complex_Type;
  with function Make (Left, Right : Real) return Complex_Type;
  with function "*" (Left, Right : Complex_Type) return Complex_Type;
  with function "+" (Left, Right : Complex_Type) return Complex_Type;
  with function "-" (Left, Right : Complex_Type) return Complex_Type;
procedure Fourier_Generic (Data     : in out Complex_Array;
                           Positive : in     Boolean := True);
```

Driver for Fast Fourier Routine

```
with Fourier_Generic;
with Complex_Generic;
with Math_Lib;
with Text_Io;
procedure Xfourier_Generic is  -- Good Ada driver for routine Fourier_Generic
  package Math is new Math_Lib (Float);
  package Complex is new Complex_Generic (Float);
  type Complex_Array is array (Positive range <>) of Complex.Complex_Type;
```

```
procedure Four1 is new Fourier_Generic (Float,
                                        Math.Sin,
                                        Complex.Complex_Type,
                                        Complex_Array,
                                        Complex.Make,
                                        Complex."*",
                                        Complex."+",
                                        Complex."-");
package Int_Io is new Text_Io.Integer_Io (integer);
package Float_Io is new Text_Io.Float_Io (Float);
Size     : constant Integer := 32;
Half_Size : constant Integer := Size / 2;
Complex_Data, Dcmp : Complex_Array (1 .. Size);
Outfile : Text_Io.File_Type;

procedure Prntft (Data : Complex_Array) is
 Length  : Integer := Data'Last - Data'First + 1;
 Line    : Integer := 0;
 Rev_Ptr : Integer;
begin
 Text_Io.New_Line;
 Text_Io.Set_Col (4);
 Text_Io.Put ("n");
 Text_Io.Set_Col (10);
 Text_Io.Put ("Real H(n)");
 Text_Io.Set_Col (22);
 Text_Io.Put ("Imag H(n)");
 Text_Io.Set_Col (38);
 Text_Io.Put ("Real H(N-n)");
 Text_Io.Set_Col (51);
 Text_Io.Put_Line ("Imag H(N-n)");
 Int_Io.Put (Line, 4);
 Text_Io.Put ("  ");
 Float_Io.Put (Complex.Real_Of (Data (1)), 5, 6, 0);
 Float_Io.Put (Complex.Imaginary_Of (Data (1)), 5, 6, 0);
 Text_Io.Put ("     ");
 Float_Io.Put (Complex.Real_Of (Data (1)), 5, 6, 0);
 Float_Io.Put (Complex.Imaginary_Of (Data (1)), 5, 6, 0);
 Text_Io.New_Line;
 for I in Data'First + 1 .. Data'Last/2 + 1 loop
  Line    := I - 1;
  Rev_Ptr := Length + 2 - I;
  Int_Io.Put (Line, 4);
  Text_Io.Put ("  ");
  Float_Io.Put (Complex.Real_Of (Data (I)), 5, 6, 0);
  Float_Io.Put (Complex.Imaginary_Of (Data (I)), 5, 6, 0);
  Text_Io.Put ("     ");
  Float_Io.Put (Complex.Real_Of (Data (Rev_Ptr)), 5, 6, 0);
  Float_Io.Put (Complex.Imaginary_Of (Data (Rev_Ptr)), 5, 6, 0);
  Text_Io.New_Line;
 end loop;
end Prntft;
```

```
begin
 Text_Io.Create (Outfile, Text_Io.Out_File, "Output.Lis");
 Text_Io.Set_Output (Outfile);
 Text_Io.Put_Line ("h(t)=real-valued even-function");
 Text_Io.Put_Line ("H(n)=H(N-n) and real?");
 for I in Complex_Data'Range loop
  Complex_Data (I) := Complex.Make
       (1.0 / ((Float (I - Half_Size - 1) / Float (Half_Size)) ** 2 + 1.0),
        0.0);
 end loop;
 Four1 (Complex_Data);
 Prntft (Complex_Data);
 Text_Io.Put_Line ("h(t)=imaginary-valued even-function");
 Text_Io.Put_Line ("H(n)=H(N-n) and imaginary?");
 for I in Complex_Data'Range loop
  Complex_Data (I) := Complex.Make
       (0.0,
        1.0 / ((Float (I - Half_Size - 1) / Float (Half_Size)) ** 2 + 1.0));
 end loop;
 Four1 (Complex_Data);
 Prntft (Complex_Data);
 Text_Io.Put_Line ("h(t)=real-valued odd-function");
 Text_Io.Put_Line ("H(n)=-H(N-n) and imaginary?");
 for I in Complex_Data'Range loop
  Complex_Data (I) := Complex.Make
       ((Float (I - Half_Size) - 1.0) / Float (Half_Size) /
        (((Float (I - Half_Size) - 1.0) / Float (Half_Size)) ** 2 + 1.0),
        0.0);
 end loop;
 Complex_Data (1) := Complex.Make (0.0, 0.0);
 Four1 (Complex_Data);
 Prntft (Complex_Data);
 Text_Io.Put_Line ("h(t)=imaginary-valued odd-function");
 Text_Io.Put_Line ("H(n)=-H(N-n) and real?");
 for I in Complex_Data'Range loop
  Complex_Data (I) := Complex.Make
       (0.0,
       (Float (I - Half_Size) - 1.0) / Float (Half_Size) /
         (((Float (I - Half_Size) - 1.0) / Float (Half_Size)) ** 2 + 1.0));
 end loop;
 Complex_Data (1) := Complex.Make (0.0, 0.0);
 Four1 (Complex_Data);
 Prntft (Complex_Data);
 -- transform, inverse-transform test
 for I in Complex_Data'Range loop
  Complex_Data (I) := Complex.Make
    (1.0/((0.5 * Float (I - Half_Size - 1) / Float (Half_Size)) ** 2 + 1.0),
     (0.25 * Float (I - Half_Size - 1) / Float (Half_Size)) *
      Math.Exp (-(0.5 * Float (I - Half_Size - 1) / Float (Half_Size))**2));
  Dcmp (I) := Complex_Data (I);
 end loop;
 Four1 (Complex_Data);
 Four1 (Complex_Data, False);
 Text_Io.New_Line;
```

```
  Text_Io.Put ("      Original Data:");
  Text_Io.Set_Col (44);
  Text_Io.Put_Line ("Double Fourier Transform:");
  Text_Io.New_Line;
  Text_Io.Put ("k     Real h(k)    Imag h(k)");
  Text_Io.Set_Col (41);
  Text_Io.Put_Line ("Real h(k)   Imag h(k)");
  Text_Io.New_Line;
  for I in Complex_Data'First .. Complex_Data'Last / 2 loop
    Int_Io.Put (I, 4);
    Float_Io.Put (Complex.Real_Of (Dcmp (I)), 7, 6, 0);
    Float_Io.Put (Complex.Imaginary_Of (Dcmp (I)), 5, 6, 0);
    Float_Io.Put (Complex.Real_Of (Complex_Data (I)) / Float (Size), 10, 6, 0);
    Float_Io.Put (Complex.Imaginary_Of (Complex_Data(I)) /
                  Float (Size), 5, 6, 0);
    Text_Io.New_Line;
  end loop;
  Text_Io.Close (Outfile);
end Xfourier_Generic;
```

REFERENCES

REFERENCES

[1] Boehm, B. *Software Engineering Economics*. New Jersey: Prentice-Hall, Inc., 1981.

[2] Giallombardo, R. "Effort and Schedule Models for Ada Software Development." MTR 11303, MITRE Corporation, Bedford, MA, May 1992.

[3] Kemerer, C. "Software Cost Estimation Models" in *Software Engineering Reference Handbook*. Surrey, U.K:, Butterworth's Scientific Limited, 1991.

[4] Press, W., S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in FORTRAN, the Art of Scientific Computing*. Second Edition, Cambridge University Press, 1992.

[5] Conversations with Software Engineering Library personnel at Computer Sciences Corporation in Greenbelt, Maryland, and estimates from Kaman Sciences personnel in Colorado Springs, Colorado.

[6] Software Productivity Consortium, Inc. "Ada Quality and Style: Guidelines for Professional Programmers." SPC-91061-CM, Version 02.01.01, December 1992.

[7] *American National Standard Programming Language FORTRAN*. American National Standards Institute, Inc., April 3, 1978.

[8] Software Productivity Consortium, Inc. "Code Counting Rules and Category Definitions/Relationships." CODE_COUNT_RULES-90010-N, Version 02.00.04, April 1991.

[9] "IEEE Standard for Software Productivity Metrics." IEEE Std 1045-1992, Institute of Electrical and Electronics Engineers, Inc., 1993.

[10] McGarry, F., and W. Agresti. "Measuring Ada for Software Development in the Software Engineering Laboratory" in *The Journal of Systems and Software*, Vol. 9, 1989, pp. 149-159.

ABBREVIATIONS

ABBREVIATIONS

DoD	Department of Defense
FFT	Fast Fourier Transform
FORTRAN	Formula Translation
IDA	Institute for Defense Analyses
KLOC	thousand lines of code
LSS	logical source statement
NASA	National Aeronautics and Space Administration
PSS	physical source statement
SDIO	Strategic Defense Initiative Organization
SEL	Software Engineering Library